ECE264 Fall 2013
Exam 3, November 20, 2013

In signing this statement, I hereby certify that the work on this exam is my own and that I have not copied the work of any other student while completing it. I understand that, if I fail to honor this agreement, I will receive a score of ZERO for this exam and will be subject to possible disciplinary action.

Signature:

You must sign here. Otherwise you will receive a **2-point** *penalty.*

This exam has 20 multiple-choice questions. For each question, select one and only one answer from the six options. If an answer requires multiple lines of code, the answer will be enclosed by { and }.

- If you do not select any answer, you will receive no point and no penalty for that question.
- If you select the correct answer, you will receive one point for that question.
- If you select an incorrect answer, you will receive 0.1 point **penalty**. The minimum score of this exam is zero. You will not receive a negative score.

It is **unlikely** that a question has two correct answers. If you think two or more answers are correct for one question, please mark so and you will receive the point if any of your selected answers is correct. You **must explain** why two or more answers are correct. You will receive no point if you do not explain.

Read the questions carefully.
Some questions have conditions and restrictions.
Please remove and return only the first sheet.

This is an *open-book, open-note* exam. You may use any book, notes, or program printouts. No personal electronic device is allowed. You may not borrow books from other students.

Three learning objectives (recursion, structure, and dynamic structure) are tested in this exam. To pass an objective, you must receive 50% or more points in the corresponding question.

Contents

1 Structure (10 points) **3**

2 Dynamic Structure and Recursion (10 points) **15**

Learning Objective 1 (Recursion)	Pass	Fail
Learning Objective 2 (Structure)	Pass	Fail
Learning Objective 3 (Dynamic Structure)	Pass	Fail

Total Score:

F means "None of the above"

Q1						
1.	A	B	C	D	E	F
2.	A	B	C	D	E	F
3.	A	B	C	D	E	F
4.	A	B	C	D	E	F
5.	A	B	C	D	E	F
6.	A	B	C	D	E	F
7.	A	B	C	D	E	F
8.	A	B	C	D	E	F
9.	A	B	C	D	E	F
10.	A	B	C	D	E	F

Q2						
1.	A	B	C	D	E	F
2.	A	B	C	D	E	F
3.	A	B	C	D	E	F
4.	A	B	C	D	E	F
5.	A	B	C	D	E	F
6.	A	B	C	D	E	F
7.	A	B	C	D	E	F
8.	A	B	C	D	E	F
9.	A	B	C	D	E	F
10.	A	B	C	D	E	F

1 Structure (10 points)

ASCII uses 8 bits to store characters. This is inefficient if only decimal digits (0, 1, 2, ..., 9) are needed. For storing one decimal digit, only 4 bits are necessary. This question asks you to implement a structure called `DecPack` because it packs two decimal digits into one byte (`unsigned char`). Each `DecPack` object has three attributes:

- `size`: the maximum number of decimal digits that can be stored in a `DecPack` object
- `used`: the actual number of decimal digits that is stored in a `DecPack` object
- `data`: an array of `unsigned char`. Each element stores **two** decimal digits. The upper (i.e., left) 4 bits stores one decimal digit and the lower (i.e., right) 4 bits stores another decimal digit.
  If the attribute `size` is an even number, the size of the array should be `size / 2`
  If the attribute `size` is an odd number, the size of the array should be `(size + 1) / 2`

When inserting a decimal digit using `DecPack_insert`, the function checks whether the `data` is full. If it is full, the size of the `data` array doubles. The old array is copied to the new array and and the memory for the old array is released. If a byte has not been used, the decimal digit uses the upper 4 bits. If the upper 4 bits of a byte is already used, the decimal digit uses the lower 4 bits.

When deleting a decimal digit using `DecPack_delete`, the function modifies `used` and returns the most recently inserted decimal digit. The digit's value must be between 0 and 9 (**not** '0' to '9'). `DecPack_delete` does **not** shrink the `data` array even if `used` is zero.

The `DecPack_print` function prints the decimal digits stored in the object. The printed decimal digits should be between '0' and '9'— if the decimal digit is 0, '0' is printed, if the decimal digit is 1, '1' is printed, and so on. The function does **not** print invisible characters.

Finally, `DecPack_destroy` releases the memory.

Please select the answers for

```
<--- FILL CODE --->
```

```
1  # include <stdio.h>
2  # include <stdlib.h>
3  # include <string.h>
4  typedef struct
5  {
6    int size; // how many digits can be stored
7    int used; // how many digits are actually stored
8    unsigned char * data; // store the digits
9    // each byte can store two digits
10 } DecPack;
11
12 // create a DecPack object with the given size
13 // sz is the maximum number of decimal digits this object can store
14 DecPack * DecPack_create(int sz)
```

```
15 {
16   // allocate memory for DecPack
17   // 1. <--- FILL CODE --->
18   /* A: */ DecPack * dp = malloc(sizeof(int));
19
20   /* B: */ DecPack * dp = malloc(sizeof(int) * sz);
21
22   /* C: */ DecPack   dp = malloc(sizeof(DecPack));
23
24   /* D: */ DecPack * dp = malloc(sizeof(DecPack));
25
26   /* E: */ DecPack * dp -> data = malloc(sizeof(unsigned char) * sz);
27
28   // check whether allocation fails
29   if (dp == NULL)
30     {
31       return NULL;
32     }
33   // initialize size to sz and used to 0
34   dp -> size = sz;
35   dp -> used = 0;
36   // allocate memory for data, each element can store two digits
37   // can store two digits
38   // allocate enough memory, no more than necessary
39   // 2. <--- FILL CODE --->
40   /* A: */ dp -> data = malloc(sizeof(unsigned char) * sz);
41
42   /* B: */ dp -> data = malloc(sizeof(int));
43
44   /* C: */ dp -> data = malloc(sizeof(DecPack));
45
46   /* D: */
47   {
48     if ((sz % 2) == 1) { sz --; }
49     dp -> data = malloc(sizeof(unsigned char) * (sz / 2));
50   }
51
52   /* E: */
53   {
54     if ((sz % 2) == 1) { sz ++; }
55     dp -> data = malloc(sizeof(unsigned char) * (sz / 2));
56   }
```

```
57   // check whether allocation fails
58   if (dp -> data == NULL)
59     {
60       free (dp);
61       return NULL;
62     }
63   // return the allocate memory
64   return dp;
65 }
66
67 // Insert a decimal digit into the DecPack object. The new digit is at
68 // the end of the sequence
69 // *** You can assume 0 <= val <= 9 ***
70 void DecPack_insert(DecPack * dp, int val)
71 {
72   // if the object is empty, do nothing
73   if (dp == NULL) { return; }
74
75   // if val < 0 or val > 9, ignore and do nothing
76   if ((val < 0) || (val > 9)) { return; }
77
78   // If the allocated memory is full, double the size and allocate
79   // memory, copy the data, release the old memory, and insert the new
80   // digit
81   int used = dp -> used;
82   if (used == dp -> size)
83     {
84       // allocate memory for double size
85       // allocate enough memory, no more than necessary
86       // Hint: each element of data can store two digits
87       // 3. <---- FILL CODE ---->
88       /* A: */
89       {
90         unsigned char * newdata =  malloc(sizeof(unsigned char) *
91                                           (dp -> size));
92         int iter;
93         (dp -> size) *= 2;
94         for (iter = 0; iter < used; iter ++)
95           {
96             newdata[iter / 2] = dp -> data[iter / 2];
97           }
98         free (dp -> data);
```

```
 99          dp -> data = newdata;
100        }
101
102        /* B: */
103        {
104          unsigned char * newdata = malloc(sizeof(unsigned char) *
105                                           (dp -> size));
106          int iter;
107          for (iter = 0; iter < used; iter ++)
108            {
109              newdata[iter / 2] = dp -> data[iter / 2];
110            }
111          free (dp -> data);
112          dp -> data = newdata;
113        }
114
115        /* C: */
116        {
117          unsigned char * newdata = malloc(sizeof(unsigned char) *
118                                           (dp -> size));
119          int iter;
120          free (dp -> data);
121          (dp -> size) *= 2;
122          dp -> data = newdata;
123        }
124
125        /* D: */
126        {
127          (dp -> size) *= 2;
128          dp -> data = malloc(sizeof(unsigned char) * (dp -> size));
129        }
130
131        /* E: */
132        {
133          unsigned char * newdata = malloc(sizeof(unsigned char) *
134                                           (dp -> size));
135          int iter;
136          free (dp -> data);
137          for (iter = 0; iter < used; iter ++)
138            {
139              newdata[iter / 2] = dp -> data[iter / 2];
140            }
```

```
141         (dp -> size) *= 2;
142         dp -> data = newdata;
143       }
144     }
145   // Hint: Make sure you do not lose the data already stored in
146   // DecPack
147
148   if ((used % 2) == 0) // it is even
149     {
150       // 4. <---- FILL CODE ---->
151       /* A: */ dp -> data[used]     = (val << 4);
152       /* B: */ dp -> data[used / 2] = (val << 4);
153       /* C: */ dp -> data[used / 2] = (val >> 4);
154       /* D: */ dp -> data[used / 2] = (val & 0XF0);
155       /* E: */ dp -> data[used]     = (val & 0X0F);
156     }
157   else // used is odd number
158     {
159       // 5. <---- FILL CODE ---->
160       /* A: */ dp -> data[used]     += val;
161       /* B: */ dp -> data[used / 2] += (val >> 4);
162       /* C: */
163       {
164         unsigned char upper = dp -> data[used / 2] & 0XF0;
165         dp -> data[used / 2] = upper + val;
166       }
167       /* D: */ dp -> data[used]     += (val & 0X80);
168       /* E: */ dp -> data[used / 2] += (val << 4);
169     }
170   (dp -> used) ++;
171 }
172
173 // delete and return the last digit in the DecPack object
174 // do not shrink the data array even if nothing is stored
175 // if the object has no value, return -1
176 int DecPack_delete(DecPack * dp)
177 {
178   // if the object is empty, do nothing
179   if (dp == NULL) { return -1; }
180   // return -1 if the DecPack object stores no data
181   if ((dp -> used) == 0) { return -1; }
182
```

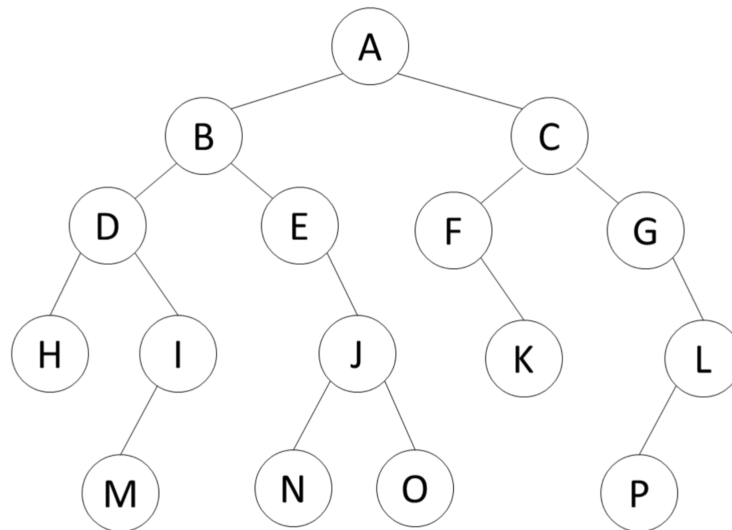
```
183   int val;
184   (dp -> used) --;
185   int used = dp -> used;
186   // Hint: make sure the returned digit is between 0 and 9
187   if ((used % 2) == 0) // it is even
188     {
189       // 6. <---- FILL CODE ---->
190       /* A: */ val = (dp -> data[used / 2]) >> 4;
191       /* B: */ val = (dp -> data[used / 2]) & 0X0F;
192       /* C: */ val = (dp -> data[used / 2]) << 4;
193       /* D: */ val = (dp -> data[used]) & 0X0F;
194       /* E: */ val = (dp -> data[used]) & 0XF0;
195     }
196   else // is odd
197     {
198       // 7. <---- FILL CODE ---->
199       /* A: */ val = (dp -> data[used / 2]) >> 4;
200       /* B: */ val = (dp -> data[used / 2]) & 0X0F;
201       /* C: */ val = (dp -> data[used / 2]) << 4;
202       /* D: */ val = (dp -> data[used]) & 0X0F;
203       /* E: */ val = (dp -> data[used / 2]) & 0XF0;
204     }
205   // return the digit
206   return val;
207 }
208
209 // print the digits stored in the object, the first inserted digit
210 // should be printed the first
211 // make sure the printed digits are between '0' and '9'
212 // This function does not print invisible characters
213 void DecPack_print(DecPack * dp)
214 {
215   // if the object is empty, do nothing
216   if (dp == NULL) { return; }
217   int iter;
218   int used = dp -> used;
219
220   // go through every digit stored in the data attribute
221   for (iter = 0; iter < used; iter ++)
222     {
223       if ((iter % 2) == 0)
224         {
```

```
225             // 8. <---- FILL CODE ---->
226             A. printf("%d", (dp-> data[iter] << 4));
227             B. printf("%d", (dp-> data[iter] >> 4));
228             C. printf("%c", (dp-> data[iter / 2]));
229             D. printf("%c", (dp-> data[iter / 2] & 0X80));
230             E. printf("%d", (dp-> data[iter / 2] >> 4));
231           }
232         else
233           {
234             // 9. <---- FILL CODE ---->
235             A. printf("%d", (dp-> data[iter / 2] >> 4));
236             B. printf("%d", (dp-> data[iter / 2] & 0X0F));
237             C. printf("%c", (dp-> data[iter / 2] & 0XF0));
238             D. printf("%c", (dp-> data[iter] & 0XFF));
239             E. printf("%d", (dp-> data[iter] << 4));
240           }
241       }
242   printf("\n");
243 }
244
245 // destroy the whole DecPack object, release all memory
246 void DecPack_destroy(DecPack * dp)
247 {
248   // if the object is empty, do nothing
249   if (dp == NULL) { return; }
250   // release the memory for the data
251   // release the memory for the object
252   // 10. <---- FILL CODE ---->
253   /* A: */
254   {
255     free ((unsigned char ) dp -> data);
256     free ((DecPack ) dp);
257   }
258   /* B: */
259   {
260     free (dp);
261     free (dp -> data);
262   }
263   /* C: */
264   {
265     free (dp -> data);
266     free (dp);
```

```
267    }
268    /* D: */
269    {
270      free (dp -> data);
271    }
272    /* E: */
273    {
274      free (dp);
275    }
276  }
277
278  int main ( int argc , char * * argv )
279  {
280    DecPack * dp = DecPack_create(5);
281    int iter;
282    for (iter = 0; iter < 21 ; iter ++)
283      {
284        DecPack_insert(dp, iter % 10);
285      }
286    DecPack_print(dp);
287    for (iter = 0; iter < 7 ; iter ++)
288      {
289        printf("delete %d\n", DecPack_delete(dp));
290      }
291    DecPack_print(dp);
292    for (iter = 0; iter < 6 ; iter ++)
293      {
294        DecPack_insert(dp, iter % 10);
295      }
296    DecPack_print(dp);
297    for (iter = 0; iter < 6 ; iter ++)
298      {
299        printf("delete %d\n", DecPack_delete(dp));
300      }
301    DecPack_print(dp);
302    DecPack_destroy(dp);
303    return EXIT_SUCCESS ;
304  }
```

2 Dynamic Structure and Recursion (10 points)

If you receive 5 points or more, you pass both learning objectives.
The question asks you to write a function
`TreeNode * findLowest(TreeNode * root, TreeNode * tn1, TreeNode * tn2)`
that finds the lowest node (i.e., the farthest from the root) that can reach both nodes `tn1` and `tn2` in a binary tree. Please be aware that this is not necessarily a binary search tree.
Consider this example



If `tn1` is I and `tn2` is E, the function returns B.
If `tn1` is N and `tn2` is O, the function returns J.
If `tn1` is F and `tn2` is H, the function returns A.
If `tn1` is B and `tn2` is I, the function returns B.
If `tn1` is K and `tn2` is P, the function returns C.
Obviously, `root` can reach any node in the tree. You can assume that neither `tn1` nor `tn2` is `NULL`. You can also assume that when `findLowest` is called outside `findLowest`, the first argument is the tree's root.
This is the algorithm. A function called
`canReach(TreeNode * p, TreeNode * q)`
determines whether it is possible to reach `q` from `p`.
`TreeNode * findLowest(TreeNode * root, TreeNode * tn1, TreeNode * tn2)` checks whether `tn1` and `tn2` are on the same side of `root`. If they are on the two different sides, the function returns `root`. If they are on the same side, the function goes down one level to the side and continues recursively.

```
1 typedef struct tnode
2 {
3   struct tnode * left;
```

```
4     struct tnode * right;
5   } TreeNode;
6
7   // Is it possible to reach q from p?
8   // return 1: yes, 0: no
9   // return 1 if q is reachable from p -> left
10  // return 1 if q is reachable from p -> right
11  // return 0 if q is not reachable from p
12
13  int canReach(TreeNode * p, TreeNode * q)
14  {
15    // return 0 if p is NULL
16    if (p == NULL) { return 0; }
17
18    // 1. <--- FILL CODE --->
19    /* A: */ while (p == q) { p = p -> next; }
20    /* B: */ if ((p -> left) == q) { return p; }
21    /* C: */ if (p == q) { return 1; }
22    /* D: */ if (p != q) { return (p -> right); }
23    /* E: */ if (p == q) { return NULL; }
24
25    // 2. <--- FILL CODE --->
26    /* A: */
27    {
28      if (canReach(p -> left, q) == 1)
29        {
30          return 1;
31        }
32    }
33
34    /* B: */
35    {
36      if (canReach(p, q) == 1)
37        {
38          return 1;
39        }
40    }
41
42    /* C: */
43    {
44      if (canReach(p -> left, q) == 0)
45        {
```

```
46          return 1;
47        }
48    }
49
50    /* D: */
51    {
52      if (canReach(p -> left, q) == 1)
53        {
54          return 0;
55        }
56    }
57
58    /* E: */
59    {
60      if (canReach(p, q) == 0)
61        {
62          return 1;
63        }
64    }
65
66    if (canReach(p -> right, q) == 1)
67      {
68        return 1;
69      }
70
71    // 3. <--- FILL CODE --->
72    /* A: */ return canReach(p, q);
73    /* B: */ return 1;
74    /* C: */ return (p == q);
75    /* D: */ return (p != q);
76    /* E: */ return 0;
77  }
78  // algorithm:
79  // if the tree is empty, it is impossible to reach either
80  // if tn1 or tn2 is the root, the answer is the the root
81  // is tn1 on the left side of root?
82  // is tn2 on the same side?
83  // if they are on the two different sides, root is answer
84  // otherwise, need to go down to that side in the tree
85  // call the function recursively at the one level lower from root
86  // Hint: the function returns an address, not an integer
87
```

```
88  TreeNode * findLowest(TreeNode * root,
89                        TreeNode * tn1, TreeNode * tn2)
90  {
91    // 4. <--- FILL CODE --->
92    /* A: */ if (root == NULL) { return 1; }
93    /* B: */ if (root == NULL) { return NULL; }
94    /* C: */ if (root != NULL) { return 1; }
95    /* D: */ if (root == NULL) { return tn1; }
96    /* E: */ if (tn1 == tn2  ) { return NULL; }
97
98    // 5. <--- FILL CODE --->
99    /* A: */ if (tn1 == tn2)                      { return root; }
100   /* B: */ if ((tn1 == root) || (tn2 == root)) { return root; }
101   /* C: */ if (tn1 == tn2)                      { return NULL; }
102   /* D: */ if ((tn1 != root) || (tn2 != NULL)) { return root; }
103   /* E: */ if (tn1 != tn2)                      { return NULL; }
104
105   int tn1left;
106   int tn2left;
107   // 6. <--- FILL CODE --->
108   /* A: */ tn1left = canReach(root, tn1);
109   /* B: */ tn1left = findLowest(root, tn1 -> left, tn2);
110   /* C: */ tn1left = findLowest(root, tn1, tn2 -> left);
111   /* D: */ tn1left = canReach(root -> left, tn1);
112   /* E: */ tn1left = (tn1 == (root -> left));
113
114   // 7. <--- FILL CODE --->
115   /* A: */ tn2left = (tn2 == (root -> right));
116   /* B: */ tn2left = findLowest(root, tn1, tn2 -> right);
117   /* C: */ tn2left = canReach(root -> left, tn2);
118   /* D: */ tn2left = findLowest(root, tn1 -> right, tn2);
119   /* E: */ tn2left = canReach(root, tn2);
120
121   if (tn1left != tn2left)
122     {
123       return root;
124     }
125
126   TreeNode * nextLevel = NULL;
127   if (tn1left == 1)
128     {
129       // 8. <--- FILL CODE --->
```

```
130         /* A: */ nextLevel = root -> right;
131         /* B: */ nextLevel = tn1  -> left;
132         /* C: */ nextLevel = tn2  -> left;
133         /* D: */ nextLevel = tn1  -> right;
134         /* E: */ nextLevel = root -> left;
135       }
136     else
137       {
138         // 9. <--- FILL CODE --->
139         /* A: */ nextLevel = root -> right;
140         /* B: */ nextLevel = root -> left;
141         /* C: */ nextLevel = tn1  -> right;
142         /* D: */ nextLevel = tn2  -> right;
143         /* E: */ nextLevel = root;
144       }
145     // 10. <--- FILL CODE --->
146     /* A: */ return (tn1 == tn2);
147     /* B: */ return canReach(root, tn1);
148     /* C: */ return findLowest(root, tn1 -> left, tn2 -> left);
149     /* D: */ return findLowest(nextLevel, tn1, tn2);
150     /* E: */
151     {
152       int val1 = canReach(nextLevel, tn1 -> left, tn2 -> left);
153       int val2 = canReach(nextLevel, tn1 -> right, tn2 -> left);
154       int val3 = canReach(nextLevel, tn1 -> left, tn2 -> right);
155       int val4 = canReach(nextLevel, tn1 -> right, tn2 -> right);
156       return (val1 + val2 + val3 + val4);
157     }
158 }
```